

# 第八章 多态

面向对象程序设计(C++)





## 8 多态

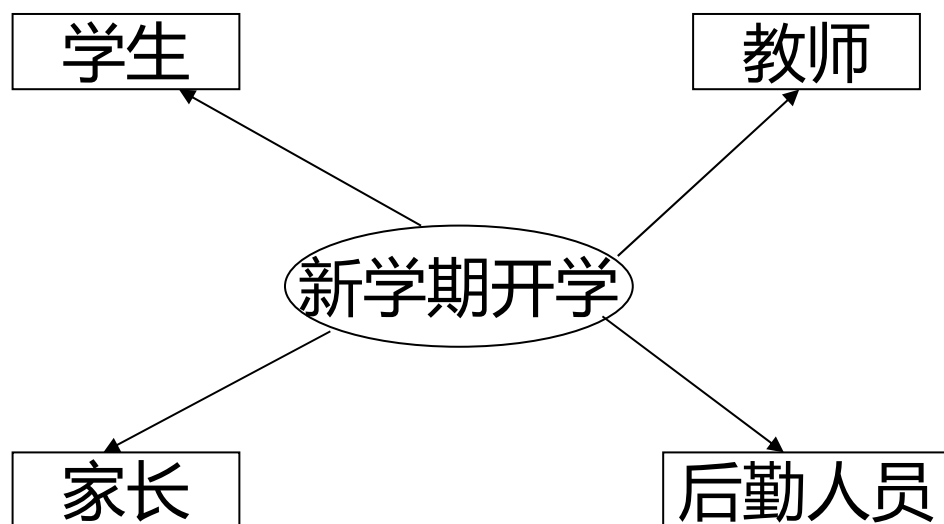
---

### 8.1 多态

### 8.2 虚函数与纯虚函数

### 8.3 抽象类

## 8.1 多态



- 向不同的对象发送同一个消息，不同的对象在接收时会产生不同的行为（方法）。每个对象可以用自己的方式去响应共同的消息  
f1（学生），f2(教师)，。。。。



# 两种多态

---

**编译时的多态：静态多态性**

**在程序编译时就能  
确定调用的是哪一个函  
数**

**运算符重载和函数重载**

**运行时的多态：动态多态性**

**在程序运行过程中动态确  
定操作所针对的对象**

**虚函数**



## 8.2 虚函数与纯虚函数

---

- 同一个类中，不能定义两个名字相同、参数个数和类型都相同的函数
- 在类的继承层次结构中，不同的层次可以出现名字相同，参数个数和类型都相同的功能不同的函数

**能否采用既能调用基类又能调用派生类的同名函数？**



## 8.2 虚函数与纯虚函数

- **虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数**
- **C++不是通过不同的对象名去调用不同派生层次中的同名函数，而是通过指针调用它们**

**虚函数是不是静态成员函数？  
静态成员函数能不能成为虚函数？**



# 虚函数的定义

**virtual <类型说明符><函数名>(<参数表>)**

- **virtual 只用在类定义的原型说明中，不能用在函数实现中。**
- **具有继承性，基类中定义了虚函数，派生类中无论是否说明，同原型函数都自动为虚函数。**
- **本质：不是重载定义而是覆盖定义。**
- **调用方式：通过基类指针或引用，执行时会根据指针指向的对象的类，决定调用哪个函数。**
- **只能用virtual声明类的成员函数，使它成为虚函数，而不能将类外的普通函数声明为虚函数**



## 虚析构函数

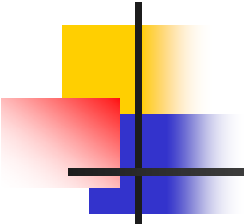
---

```
class Base1
{
public:
    ~Base1(){cout<<"~Base1()"<<endl;}
};
```

```
class Derived1:public Base1
{
public:
    ~Derived1(){cout<<"~Derived1"<<endl;}
};
```

```
class Base2
{
public:
    virtual ~Base2(){cout<<"~Base2()"<<endl;}
};
```

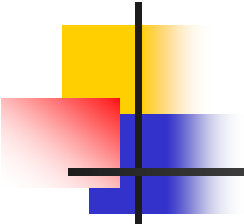




---

```
class Derived2: public Base2  
{  
public:  
    ~Derived2(){cout<<"Derived2()"<<endl;}  
};
```

```
void main()  
{  
    Base1 *bp=new Derived1;  
    delete bp;  
    Base2 *b2p=new Derived2;  
    delete b2p;  
}
```



```
class Derived2: public Base2
{
public:
    ~Derived2(){cout<<"Derived2()"<<endl;}
};
```

```
int main()
{
    Base1 *bp=new Derived1;
    delete bp;
    Base2 *b2p=new Derived2;
    delete b2p;
}
```

```
~Base1()
Derived2()
~Base2()
```

**一个类有虚函数，那么它的析构函数也应该是虚的**



# 纯虚函数的定义

---

**virtual 函数类型 函数名(参数表) = 0**

- **纯虚函数是一个在基类中说明的虚函数，在基类中没有定义具体的操作内容，要求派生类根据需要定义自己的版本**



## 8.3 抽象类

---

- **定义：不用来定义对象而只作为一种基本类型用作继承的类**
- **带有纯虚函数的类是抽象类 //无法实现**
- **抽象类只能作为基类使用，不能实例化。但是可以声明抽象类的**指针**和**引用**，指向并访问派生类的对象**



## 例子

---

```
class Shape
{
    protected:
        double x,y;

    public:
        void set(double i, double j)
        { x=i; y=j; }
        virtual void area()=0;
};
```

这个类本身是否存在实例？



## 例子

---

```
class Triangle: public Shape
{
    public:
        void area()
        { cout<< "三角形面积:  " <<0.5*x*y<<endl;  }
};
```

```
class Rectangle: public Shape
{
    public:
        void area()
        { cout<<"矩形面积:  " <<x*y<<endl;  }
};
```



## 例子

---

```
int main()
{
    Shape *p; // 基类指针
    ???
    p->area();
    ???
    p->area();
}
```



## 例子

---

```
class B{
    public:
        virtual void m1() {}
        virtual void m2() {}
};
class D: public B{
    public:
        virtual void m1() {}
};
```

**B::m1 0x7723    D::m1 0x99a7 (虚函数表vtable)**  
**B::m2 0x23b4    D::m2 0x23b4**





## 例子

---

```
int main(){  
    B b1;  
    D d1;  
    B* p;  
    p->m1(); //查表: p指向b1,...; p指向d1,...  
}
```

**虚成员函数表需要额外空间+额外查询时间，可能影响效率**



## 例子：通过引用实现多态

---

```
class person
{
    virtual void all_info() {}
};
class student: public person
{
    virtual void all_info() {}
};
int main()
{
    student tom(...);
    person& pr=tom;
    tom.all_info(); // 通过成员选择符 “.” 使用引用
}
```



## 程序理解1

---

```
class B{
    public:
        void m() {cout<< "B::m" <<endl;}
};
class D: public B{
    public:
        void m() {cout<< "D::m" <<endl;}
};
int main(){
    B* p;
    p=new D;
    p->m();
    return 0;
}
```

程序输出是什么？



# 程序理解1

---

```
class B{
    public:
        void m() {cout<< "B::m" <<endl;}
};
class D: public B{
    public:
        void m() {cout<< "D::m" <<endl;}
};
int main(){
    B* p;
    p=new D;
    p->m();
    return 0;
}
```

**B::m(编译时绑定, 根据指针类型)**



## 程序理解2

---

```
class A{
    public:
        virtual void f()=0;
};
class X: public A{
    public:
        virtual void f() {}
};
class Y: public A {};
int main(){
    A a1;
    X x1;
    Y y1;
    return 0;
}  哪些初始化代码是错误的?
```



## 程序理解2

---

```
class A{
    public:
        virtual void f()=0;
};
class X: public A{
    public:
        virtual void f() {}
};
class Y: public A {};
int main(){
    A a1; //error
    X x1;
    Y y1; //error
    return 0;
} 纯虚函数没有被重写，则仍是抽象类！
```



## 程序理解3

---

```
class Base{  
public:  
    virtual test(int a) {};  
};
```

```
class Child: public Base{  
public:  
    test(int a) const {};  
};
```

**实质是不同的函数，所以屏蔽了基类的虚函数！换句话说，Child中的test函数没有绑定Base中的test函数！为什么？**



## 程序理解3

---

```
class Base{  
public:  
    void test(int a);  
    void test(const int a); //error, 不属于重载  
}
```

```
class Base{  
public:  
    void test(int &a);  
    void test(const int &a); //right, 属于重载, 实质是不同的函数  
}
```

```
class Base{  
public:  
    void test(int a);  
    void test(int a) const; //right, 同上  
};
```





## 程序理解3

---

```
class Base{  
public:  
    virtual test(int a) {};  
};
```

```
class Child: public Base{  
public:  
    test(int a) override {};  
};
```

**表明test(int a)是重写虚函数，让编译器帮助查找错误！**



## 类型转换static\_cast

---

```
class B{ ... //没有函数m() };  
class D: public B{  
    public:  
        void m() {}  
};  
int main(){  
    D* p; //error  
    p=static_cast<D*>(new B);  
    p->m(); //error  
    return 0;  
}
```

**编译无错，但运行出错！**

**Upcast: 基类指针（自动）指向派生类；**  
**Downcast: 派生类指针指向基类不明智！**



## 类型转换dynamic\_cast

**dynamic\_cast仅对多态类型有效!**

**格式: dynamic\_cast<指针、引用>(多态类型)**

```
class B{ ... //没有函数m(), 至少有一个虚函数};  
class D: public B{  
    public:  
        void m() {}  
};  
int main(){  
    D* p=dynamic_cast<D*>(new B);  
    if (p)    //如果类型转换安全  
        p->m();  
    else  
        cerr<< "Not safe..." <<endl;  
}
```

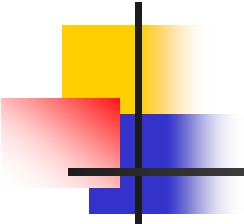


## 双分派

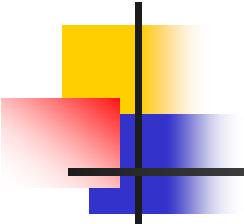
---

```
class Problem  
{  
public:  
    Problem(){}  
    virtual ~Problem(){}  
};
```

```
class SpecialProblem: public Problem  
{  
public:  
    SpecialProblem(){}  
    ~SpecialProblem(){}  
};
```



```
class Supporter
{
public:
    Supporter(){}  
    virtual ~Supporter(){}  
  
    virtual void solve(Problem &p)
{
    std::cout << "一级支持解决一般问题" << std::endl;
}
    virtual void solve(SpecialProblem &sp)
{
    std::cout << "一级支持解决特殊问题" << std::endl;
}
};
```



```
class SeniorSupporter: public Supporter
{
public:
    SeniorSupporter(){}
    ~SeniorSupporter(){}

    void solve(Problem &p)
    {
        std::cout << "资深支持解决一般问题" << std::endl;
    }
    void solve(SpecialProblem &sp)
    {
        std::cout << "资深支持解决特殊问题" << std::endl;
    }
};
```



```
int main()
```

```
{
```

```
    Problem *p=new Problem();
```

```
    Problem *sp=new SpecialProblem();
```

```
    Supporter *s=new SeniorSupporter();
```

```
    s->solve(*p);
```

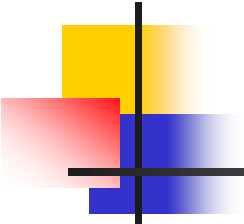
```
    s->solve(*sp);
```



函数重载?

```
    return 0;
```

```
}
```



---

```
int main()
{
    Problem *p=new Problem();
    Problem *sp=new SpecialProblem();
    Supporter *s=new SeniorSupporter();

    s->solve(*p);
    s->solve(*sp);

    return 0;
}
```

**资深支持解决一般问题  
资深支持解决一般问题**



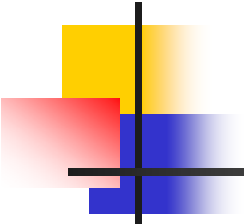


# 解决办法

---

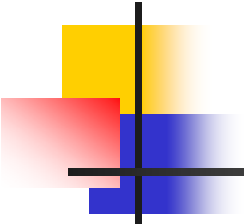
```
class Problem
{
public:
    Problem(){}
    virtual ~Problem(){}
    virtual void solve(Supporter *s)
    {
        s->solve(*this);
    }
};
```

**函数调用时将自身传入**



---

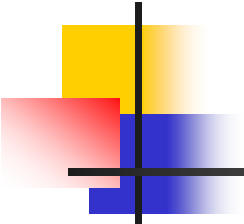
```
class SpecialProblem:public Problem  
{  
public:  
    SpecialProblem(){}  
    ~SpecialProblem(){}  
    void solve(Supporter *s)  
    {  
        s->solve(*this);  
    }  
  
};
```



```
int main()
{
    Problem *p=new Problem();
    Problem *sp=new SpecialProblem();
    Supporter *s=new SeniorSupporter();
    p->solve(s);
    sp->solve(s);

    return 0;
}
```

**两次动态分配**



```
int main()
{
    Problem *p=new Problem();
    Problem *sp=new SpecialProblem();
    Supporter *s=new SeniorSupporter();
    p->solve(s);
    sp->solve(s);

    return 0;
}
```

资深支持解决一般问题  
资深支持解决特殊问题